

Semafori Posix

Laboratorio Sistemi Operativi

Aniello Castiglione

Email: aniello.castiglione@uniparthenope.it

Semafori Posix e System V

- Un semaforo è una primitiva usata per fornire un meccanismo di sincronizzazione tra vari processi o vari thread di un processo
- Consideriamo innanzitutto un **semaforo binario**: un semaforo che può assumere solo i valori 0 (bloccato) e 1 (sbloccato)
- Possiamo considerare tre tipologie di semafori
 - I **semafori Posix** (non mantenuti nel kernel) possono essere di due tipi
 - Basati su nome ed identificati da nomi Posix per IPC
 - Basati su memoria e memorizzati in memoria condivisa
 - I **semafori nella versione System V** (mantenuti all'interno del kernel)

Operazioni sui semafori

- Un processo può eseguire 3 operazioni sui semafori
 - **Creazione**: richiede di inizializzare anche il valore di partenza (0 o 1 per i semafori binari)
 - **wait** (o anche **P**): testa il valore del semaforo, si blocca se il valore ≤ 0 , oppure decrementa il valore del semaforo una volta che risulta > 0

```
while (semaphore_value <= 0)
    ; /* wait, ovvero blocca il processo */
semaphore_value--;
```

E' necessario che il test del valore del semaforo ed il successivo decremento siano fatti con operazioni atomiche rispetto agli altri processi che accedono al semaforo

Operazioni sui semafori (2)

- **post** (o anche **V** o **Signal**) : incrementa il valore del semaforo
 - Se un processo è in attesa che il valore del semaforo sia >0 , allora tale processo può essere risvegliato
 - Anche in questo caso è richiesto che l'operazione di incremento sia atomica

Semaforo contatore

- Il concetto di semaforo contatore generalizza quello di semaforo binario
 - È un semaforo il cui valore varia tra 0 e qualche valore limite (almeno **32767** per i semafori Posix)
 - Si utilizzano per contare le istanze disponibili di una qualche risorsa. Il valore indica il numero di risorse disponibili
 - Anche in questo caso, l'operazione **wait** aspetta che il valore del semaforo sia maggiore di 0 e poi decrementa tale valore
 - L'operazione **post** incrementa il valore del semaforo e risveglia un processo in attesa che il valore del semaforo sia > 0

Semaforo binario e mutua esclusione

- Un semaforo binario può essere usato per la mutua esclusione come un mutex:

<code>inizializza il mutex;</code>	<code>inizializza il semaforo a 1;</code>
<code>pthread_mutex_lock(&mutex);</code>	<code>sem_wait(&sem);</code>
<code> <i>regione critica;</i></code>	<code> <i>regione critica;</i></code>
<code>pthread_mutex_unlock(&mutex);</code>	<code>sem_post(&sem);</code>

- Inizializziamo il semaforo ad 1. La chiamata a `sem_wait()` attende che il valore sia >0 e poi decrementa il valore
- La chiamata `sem_post()` incrementa il valore (da 0 a 1) e risveglia un thread o processo bloccato da una `sem_wait()` sullo stesso semaforo

Semaforo binario e mutua esclusione

- Osservazione: sebbene possa essere usato come un mutex, un semaforo ha una caratteristica non disponibile con i mutex
 - un mutex deve sempre essere sbloccato dal thread che lo ha bloccato
 - Un'operazione **post** su di un semaforo non deve essere necessariamente effettuata dallo stesso thread che ha effettuato una **wait** sul semaforo
- Mostriamo questo punto considerando un versione semplificata del problema del produttore e del consumatore (con buffer condiviso che, per semplicità, assumiamo contenga **un solo elemento**) usando due semafori binari

Esempio

Produttore

```
inizializza semaforo get a 0;  
inizializza semaforo put a 1;  
  
for( ; ; ) {  
    sem_wait(&put);  
    inserimento dati nel buffer;  
    sem_post(&get);  
}
```

Consumatore

```
for( ; ; ) {  
    sem_wait(&get);  
    elabora dati dal buffer;  
    sem_post(&put);  
}
```

- Il semaforo **put** controlla se il produttore può porre un elemento nel buffer condiviso
- Il semaforo **get** controlla se il consumatore può rimuovere un elemento dal buffer condiviso

Esempio (cont.)

- I passi che si possono verificare in esecuzione sono:
 1. Il produttore inizializza il buffer e i due semafori
 2. Assumiamo, poi, che venga eseguito il consumatore. Esso si blocca nella sua chiamata a `sem_wait()` poiché il valore di `get` è 0
 3. Dopo qualche istante, inizia il produttore. Quando chiama `sem_wait()`, il valore di `put` è decrementato da 1 a 0, e il produttore pone un elemento nel buffer
 - Poi invoca una `sem_post()` per incrementare il valore di `get` da 0 a 1. Poiché c'è un thread bloccato sul semaforo `get`, in attesa che il valore divenga positivo, il thread è marcato come pronto per l'esecuzione
 - Assumiamo che il produttore continui l'esecuzione. Allora il produttore si blocca nella sua chiamata a `sem_wait()` in cima al ciclo `for`, perché il valore di `put` è 0. Il produttore deve attendere fino a che il consumatore svuoti il buffer

Esempio (cont.)

4. Il consumatore ritorna dalla sua chiamata a `sem_wait()`, il che decrementa il valore del semaforo `get` da 1 a 0. Elabora l'elemento nel buffer e chiama `sem_post()` che incrementa il valore di `put` ad 0 a 1. Poiché un thread è bloccato su questo semaforo (il produttore), in attesa che il valore sia >0 , il thread viene marcato come pronto per l'esecuzione. Ma assumiamo che il consumatore continui l'esecuzione. Il consumatore si blocca nella sua chiamata a `sem_wait()`, in cima al ciclo for, poiché il valore di `get` è 0
5. Il produttore ritorna dalla sua chiamata a `sem_wait()`, pone un elemento nel buffer e lo schema si ripete

Esempio (cont.)

- Abbiamo assunto che ogni volta che è chiamata `sem_post()`, anche se un processo era in attesa e marcato come pronto per l'esecuzione, il chiamante continua l'esecuzione
 - Se il chiamante continua l'esecuzione o se il thread appena divenuto pronto viene eseguito non importa ai fini della sincronizzazione

Differenze tra meccanismi di sincronizzazione

- Possiamo elencare tre differenze tra i **semafori**, i **mutex** e le **variabili condizione**:
 1. Un **mutex** deve sempre essere sbloccato dal thread che lo ha bloccato, mentre un'operazione **post** su di un **semaforo** non deve essere necessariamente effettuata dallo stesso thread che ha invocato l'operazione **wait** sul semaforo
 2. Un **mutex** può essere nello stato bloccato o sbloccato (uno stato binario simile al semaforo binario)
 3. Poiché un **semaforo** ha uno stato associato con esso (il contatore), un'operazione **post** viene sempre ricordata
 - Quando una **variabile condizione** è segnalata, se nessun thread è in attesa per essa, il segnale viene perso

Funzioni per i semafori Posix

- Posix fornisce due tipi di semafori: con *nome* e *basati su memoria*

Semafori con nome

`sem_open()`

Semafori basati su memoria

`sem_init()`

`sem_wait()`
`sem_trywait()`
`sem_post()`
`sem_getvalue()`

`sem_close()`
`sem_unlink()`

`sem_destroy()`

Semafori Posix basati su nome

Creazione di un semaforo

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, mode_t
                mode, int value);
/* Restituisce un puntatore al semaforo se OK, SEM_FAILED in caso
   di errore */
```

- La funzione `sem_open()` crea un nuovo semaforo con nome o apre un semaforo esistente. Un semaforo con nome può sempre essere usato per sincronizzare sia i thread che i processi
- Il primo argomento è un nome di **IPC Posix** che può essere o meno un pathname reale nel filesystem
- L'argomento *oflag* può essere `0`, `O_CREAT` oppure `O_CREAT|O_EXCL`
 - Se è specificato `O_CREAT`, sono richiesti il terzo (bit di permesso) ed il quarto (valore iniziale del semaforo) argomento
 - Il valore iniziale non può superare `SEM_VALUE_MAX` (in `limits.h`) che deve essere almeno `32767`. I semafori binari hanno solitamente un valore iniziale di 1 mentre i semafori contatore hanno un valore iniziale maggiore di 1

Creazione di un semaforo

- Se è specificato `O_CREAT` (senza `O_EXCL`), il semaforo è inizializzato solo se non esiste già. Specificare `O_CREAT` su un semaforo esistente non è un errore. Specificare `O_CREAT|O_EXCL` su un semaforo esistente è un errore
- Il valore di ritorno è un puntatore al tipo `sem_t`
- Il puntatore è utilizzato come argomento alle altre funzioni `sem_close()`, `sem_wait()`, `sem_trywait()`, `sem_post()` e `sem_getvalue()`

Chiusura di un semaforo

```
#include <semaphore.h>
int sem_close(sem_t *sem) ;
    /* Restituisce 0 se OK, -1 in caso di errore */
```

- Un semaforo aperto con `sem_open()` è chiuso con `sem_close()`
- Questa operazione si verifica automaticamente anche quando viene terminato (volontariamente o involontariamente) un processo per qualsiasi semaforo con nome aperto
- La chiusura di un semaforo non lo rimuove dal sistema. I nomi di semafori Posix sono **persistenti almeno a livello kernel**: mantengono il proprio valore anche se nessun processo ha il semaforo aperto

Persistenza degli oggetti di IPC

- La persistenza di un oggetto di un qualsiasi tipo di IPC è definita come la quantità di tempo in cui l'oggetto di quel tipo esiste
 - **Persistenza di processo**: l'oggetto esiste fino a che l'ultimo processo che mantiene l'oggetto aperto lo chiude (*pipe e FIFO*)
 - **Persistenza a livello kernel**: l'oggetto esiste fino al reboot o fino a che l'oggetto è cancellato esplicitamente (*semafori SV*). I semafori Posix e la memoria condivisa Posix devono essere persistenti almeno al livello kernel, ma possono anche essere persistenti a livello di file system, a seconda dell'implementazione
 - **Persistenza a livello di file system**: l'oggetto esiste fino a che esso è esplicitamente cancellato. L'oggetto mantiene il proprio valore anche se il kernel effettua un reboot (*semafori e memoria condivisa Posix* possono avere tale proprietà)

Rimozione di un semaforo

- Un semaforo con nome è rimosso dal sistema con la funzione `sem_unlink()`

```
#include <semaphore.h>
```

```
int sem_unlink(const char *name);
```

```
/* Restituisce 0 se OK, -1 in caso di errore */
```

- I semafori hanno un contatore di riferimento che tiene traccia di quante volte sono aperti correntemente
 - Il nome del semaforo può essere rimosso dal filesystem mentre il suo contatore è maggiore di zero
 - La sua rimozione effettiva, però, non avviene fino a che non si verifica l'ultima `sem_close()`

sem_wait e sem_trywait

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

```
int sem_trywait(sem_t *sem);
```

```
/* Restituisce 0 se OK, -1 in caso di errore */
```

- La funzione `sem_wait()` testa il valore del semaforo specificato, e se il valore è maggiore di 0, il valore è decrementato e la funzione ritorna immediatamente
- Se il valore è 0 quando la funzione è invocata, il thread chiamante è messo in attesa (sleep) fino a che il valore del semaforo diventa maggiore di 0, momento in cui il valore sarà decrementato e la funzione ritorna
- Le operazioni “testa e decrementa” devono essere atomiche rispetto agli altri thread che accedono al semaforo
- La differenza tra `sem_wait()` e `sem_trywait()` è che la seconda non pone il thread in attesa nel caso in cui il valore del semaforo sia 0. Invece, viene restituito l’errore `EAGAIN`
- `sem_wait()` può ritornare prima se interrotta da un segnale (errore `EINTR`)

sem_post e sem_getvalue

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

```
int sem_getvalue(sem_t *sem, int *valp);
```

```
/* Restituisce 0 se OK, -1 in caso di errore */
```

- **sem_post()** incrementa il valore del semaforo di 1 e risveglia qualsiasi thread in attesa che il suo valore diventi positivo
- **sem_getvalue()** restituisce il valore corrente del semaforo nell'intero puntato da *valp*. Se il semaforo è bloccato, **Posix.1-2001** indica due possibilità:
 - il valore restituito è 0 (soluzione adottata da Linux e OpenSolaris)
 - o un numero negativo il cui valore assoluto è il numero di thread in attesa che il semaforo sia sbloccato

Esempio

- Vediamo qualche esempio di programma che opera sui semafori Posix con nome
- Poiché i semafori Posix hanno almeno persistenza a livello del kernel li possiamo usare attraverso programmi multipli
- Il programma che segue (*semcreate*) crea un semaforo e consente di specificare due opzioni: *-e* per specificare una creazione esclusiva e *-i* per specificare un valore iniziale (diverso da 1, quello di default)

```

#define FILE_MODE S_IRUSR|S_IWUSR
int main(int argc, char **argv)
{
    int c, flags;
    sem_t *sem;
    unsigned int value;
    flags = O_CREAT;
    value = 1;
    while ( (c = getopt(argc, argv, "ei:")) != -1) {
        switch (c) {
            case 'e':
                flags |= O_EXCL;
                break;
            case 'i':
                value = atoi(optarg);
                break;
        }
    }
    if (optind != argc-1) {
        fprintf(stderr, "usage: semcreate [-e] [-i initialvalue] <name>");
        exit(-1);
    }
    sem = sem_open(argv[optind], flags, FILE_MODE, value);
    sem_close(sem);
    exit(0);
}

```

semcreate

Funzione getopt

```
#include <unistd.h>
```

```
int getopt(int argc, char *const argv[], const  
char *optstring);  
extern char *optarg;  
extern int optind;
```

- La funzione getopt effettua il parsing incrementale degli argomenti da riga di comando (argv) e restituisce il carattere d'opzione noto successivo
- L'opzione è nota se è stata specificata nella stringa dei caratteri d'opzione accettati, **optstring**
 - In *optstring*, se un carattere è seguito dai due punti (:), l'opzione richiede un argomento
- La variabile *optind* è l'indice del successivo elemento di argv[] da elaborare
 - Inizializzato ad 1 dal sistema ed aggiornato da getopt()

Funzione getopt (cont.)

- La funzione *getopt()* restituisce il prossimo carattere d'opzione (se trovato) da *argv* che corrisponde ad un carattere in *optstring*
- Se l'opzione richiede un argomento, *getopt()* imposterà *optarg* in modo da puntare all'argomento opzione

semcreate

- Dal momento che si usa sempre **O_CREAT** dobbiamo invocare **sem_open()** con quattro argomenti
 - I due argomenti finali, però, sono usati da **sem_open()** solo se il semaforo non esiste già

semunlink

```
#include    ...

int main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "usage: semunlink <name>");
        exit(-1);
    }

    sem_unlink(argv[1]);

    exit(0);
}
```

semgetvalue

```
#include      ...
int
main(int argc, char **argv)
{
    sem_t      *sem;
    int        val;

    if (argc != 2) ){
        fprintf(stderr, "usage: semgetvalue <name>");
        exit(-1);
    }
    sem = sem_open(argv[1], 0);
    sem_getvalue(sem, &val);
    printf("value = %d\n", val);

    exit(0);
}
```

semgetvalue

- Apre un semaforo, preleva il suo valore corrente ed infine ne stampa il valore
- Quando si sta per aprire un semaforo che esiste già, il secondo argomento di `sem_open()` è 0: non viene specificato `O_CREAT`

semwait

```
#include      ...

int
main(int argc, char **argv)
{
    sem_t      *sem;
    int        val;

    if (argc != 2){
        fprintf(stderr, "usage: semwait <name>");
        exit(-1);}
    sem = sem_open(argv[1], 0);
    sem_wait(sem);
    sem_getvalue(sem, &val);
    printf("pid %ld has semaphore, value = %d\n", (long)
    getpid(), val);

    pause(); /* bloccato fino a terminazione da utente */
    exit(0);
}
```

semwait

- Apre un semaforo
- Chiama `sem_wait()` (che si bloccherà se il valore corrente del semaforo è minore o uguale di 0 e dopo decrementa il valore del semaforo)
- Preleva e stampa il valore del semaforo e poi si blocca per sempre chiamando una `pause()`

sempost

```
#include      ...

int
main(int argc, char **argv)
{
    sem_t      *sem;
    int        val;

    if (argc != 2){
        fprintf(stderr, "usage: sempost <name>");
        exit(-1);
    }
    sem = sem_open(argv[1], 0);
    sem_post(sem);
    sem_getvalue(sem, &val);
    printf("value = %d\n", val);

    exit(0);
}
```


sempost

- Invia un post ad un semaforo (ne incrementa il valore di 1) e successivamente preleva e stampa il valore del semaforo

Semafori Posix sotto Linux e OpenSolaris

- I semafori Posix non sono disponibili in tutti i sistemi
- Per compilare un sorgente che usa semafori Posix è necessario includere la libreria real-time *librt* o *libpthread*
 - *gcc nomefile.c -lrt*
- Il nome del semaforo è specificato con */nomesemaforo*
 - Linux
 - Il nome è creato nel file system e montato in */dev/shm* con il nome *sem.nomesemaforo*
 - OpenSolaris
 - Il nome è creato nel file system e montato in */tmp* con il nome *.SEMDnomesemaforo*

Esempio di esecuzione su Linux e OpenSolaris

```
$ semcreate /test1
```

- Linux: creato `sem.test1` in `/dev/shm`
- OpenSolaris: creato `.SEMDtest1` in `/tmp`

```
$ semgetvalue /test1
```

```
value = 1
```

- Il kernel non effettua un post ad un semaforo in modo automatico quando il processo che mantiene il semaforo occupato termina senza rilasciarlo:

```
$ semwait /test1
```

```
Pid 4133 has semaphore, value = 0
```

```
^?
```

```
$ semgetvalue /test1
```

```
value = 0
```

Esempio di esecuzione su Linux

```
$ semgetvalue /test1
```

```
value = 0
```

```
$ semwait /test1 &
```

```
[1] 4257
```

```
$ semgetvalue /test1
```

```
value = 0
```

```
$ semwait /test1 &
```

```
[2] 4263
```

```
$ semgetvalue /test1
```

```
value = 0
```

```
$ sempost /test1
```

```
pid 4257 has semaphore, value =0
```

output del programma semwait

```
value = 0
```

```
$ sempost /test1
```

```
pid 4263 has semaphore, value = 0
```

output del programma semwait

```
value = 0
```

```
$
```

ancora 0 dall'esempio di prima
parte in bckgrn e si blocca

non usa valori negativi
un altro processo in bckgrn che si blocca

due processi in attesa, ma il valore resta 0

Esempio di esecuzione su Digital Unix

- Verifichiamo una diversa implementazione rispetto a Linux e OpenSolaris
- Creiamo un semaforo con nome e ne stampiamo il suo valore di default

```
$ semcreate /tmp/test2
```

```
$ ls -l /tmp/test2
```

```
-rw-r- -r-- 1 username groupname /tmp/test2
```

```
$ semgetvalue /temp/test2
```

```
value = 1
```

- Il sistema crea un file nel filesystem che corrisponde al nome che abbiamo specificato per il semaforo

Esempio di esecuzione

- Aspettiamo il semaforo e terminiamo (abort) il programma che mantiene il semaforo bloccato

```
$ semwait /temp/test2
```

```
Pid 9702 has semaphore, value = 0
```

```
^?
```

```
$ semgetvalue /tmp/test2
```

```
value = 0
```

- Questo esempio mostra due caratteristiche menzionate in precedenza
 1. Il valore di un semaforo è persistente nel kernel, ovvero il valore 1 è mantenuto dal kernel da quando il semaforo è stato creato, anche se nessun programma ha il semaforo aperto in quel momento
 2. Quando terminiamo *semwait* che mantiene il semaforo bloccato, il valore del semaforo non cambia. Cioè, il semaforo non è sbloccato dal kernel quando il processo che mantiene il blocco termina senza sbloccare

Esempio di esecuzione

- Mostriamo che questa implementazione usa valori negativi del contatore per indicare il numero di processi in attesa che il semaforo si sblocchi

```
$ semgetvalue /tmp/test2
```

```
value = 0
```

ancora 0 dall'esempio di prima

```
$ semwait /tmp/test2 &
```

parte in bckgrn e si blocca

```
[1] 9718
```

```
$ semgetvalue /tmp/test2
```

```
value = -1
```

un processo in attesa

```
$ semwait /tmp/test2 &
```

un altro processo in bckgrn
che si blocca

```
[2] 9727
```

```
$ semgetvalue /tmp/test2
```

due processi in attesa

```
value = -2
```

Esempio di esecuzione

```
$ sempost /tmp/test2
```

```
value = -1
```

```
pid 9718 has the semaphore, value = -1
```

```
$ sempost /tmp/test2
```

```
value = 0
```

```
pid 9727 has semaphore, value = 0
```

- Quando il valore è -2 ed eseguiamo sempost, il valore è incrementato a -1 ed uno dei processi bloccati nella chiamata a **sem_wait()** ritorna

Il problema produttore-consumatore

- Introducendo mutex e variabili di condizione, abbiamo visto alcune soluzioni possibili per sincronizzare l'attività di n thread produttori che inserivano elementi in un buffer condiviso ed elaborati da un singolo thread consumatore
 1. Nella prima soluzione, il consumatore era avviato solo dopo che gli n produttori avevano completato il loro compito. Il problema si risolveva con un unico mutex
 2. Nella soluzione successiva, il consumatore era avviato prima che gli n produttori finissero il loro compito. Ciò ha richiesto un mutex per sincronizzare i produttori ed una variabile di condizione (con relativo mutex) per sincronizzare il consumatore con i produttori

Problema Produttore-Consumatore

- Estendiamo il problema del produttore e del consumatore visto per i mutex e le variabili condizione usando un buffer condiviso circolare:
 - Dopo che il produttore ha riempito l'ultima entrata (`buff[NBUFF-1]`), ritorna indietro e riempie la prima entrata (`buff[0]`). Il consumatore fa lo stesso
 - Ciò aggiunge un ulteriore problema di sincronizzazione poiché il produttore non deve superare il consumatore
 - Assumiamo che produttore e consumatore siano thread (possono essere anche processi)

Problema Produttore-Consumatore

- Tre condizioni devono essere rispettate quando il buffer è circolare:
 1. Il consumatore non può cercare di rimuovere un elemento dal buffer quando il buffer è vuoto
 2. Il produttore non può provare a inserire un elemento quando il buffer è pieno
 3. Le variabili condivise possono descrivere lo stato del buffer (indici, contatori ...), per cui tutti gli accessi al buffer del produttore e del consumatore devono essere protetti per evitare race condition

Problema Produttore-Consumatore

- La soluzione adotta tre tipi differenti di semafori:
 1. Un semaforo binario chiamato *mutex* protegge le regioni critiche: l'inserimento di un elemento nel buffer (produttore) e la rimozione di un elemento dal buffer (consumatore). Il semaforo binario usato come *mutex* è inizializzato a 1
 2. Un semaforo contatore chiamato *nempty* conta il numero di posti vuoti nel buffer. Questo semaforo è inizializzato al numero di locazioni del buffer (*NBUFF*)
 3. Un semaforo contatore *nstored* conta il numero di locazioni occupate del buffer. Questo semaforo è inizializzato a 0, poiché all'inizio il buffer è vuoto

Problema Produttore-Consumatore

- In questo esempio, il produttore memorizza gli interi tra **0** e **nitems** nel buffer (`buff[0]=0, buff[1]=1, ...`), usando il buffer come buffer circolare
- Il consumatore prende gli interi dal buffer e verifica che essi siano corretti, stampando eventuali errori sullo standard output

Variabili globali

```
#include    ...

#define     NBUFF      10
#define     SEM_MUTEX   "/mutex"
#define     SEM_NEMPTY  "/nempty"
#define     SEM_NSTORED "/nstored"

int  nitems; /* sola lettura per prod. e cons. */
struct {      /* dati condivisi da prod. e cons. */
    int buff[NBUFF];
    sem_t      *mutex, *nempty, *nstored;
} shared;

void *produce(void *), *consume(void *);
```

Variabili globali

- Il buffer contenente **NBUFF** elementi è condiviso dai due thread così come i puntatori ai semafori
- Raggruppiamo questi elementi in una struttura per evidenziare che i semafori sono usati per sincronizzare l'accesso al buffer

main

```
int
main(int argc, char **argv)
{
    pthread_t    tid_produce, tid_consume;
    if (argc != 2)
        {printf("usage: prodcons1 <#items>");exit(-1);}
    nitems = atoi(argv[1]);
        /* crea i tre semafori */
    shared.mutex = sem_open(SEM_MUTEX, O_CREAT | O_EXCL, FILE_MODE, 1);
    shared.nempty = sem_open(SEM_NEMPTY, O_CREAT | O_EXCL, FILE_MODE,
NBUFF);
    shared.nstored = sem_open(SEM_NSTORED, O_CREAT | O_EXCL, FILE_MODE,
0);
        /* crea un thread produttore ed un thread consumatore */
    pthread_create(&tid_produce, NULL, produce, NULL);
    pthread_create(&tid_consume, NULL, consume, NULL);
        /* attende i due thread */
    pthread_join(tid_produce, NULL);
    pthread_join(tid_consume, NULL);
        /* rimuove i semafori */
    sem_unlink(SEM_MUTEX);sem_unlink(SEM_NEMPTY);sem_unlink(SEM_NSTORED);
    exit(0);
}
```


main

- Sono creati tre semafori
- Ci assicuriamo che i semafori siano correttamente inizializzati usando il flag **O_EXCL** (che restituisce un errore se il semaforo già esiste)
- Creiamo due thread senza passare alcun argomento ad essi

Produttore

```
void *produce(void *arg)
{
    int          i;

    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nempty);
        /* attende almeno un posto vuoto */
        sem_wait(shared.mutex);
        shared.buff[i % NBUFF] = i;
        /* memorizza i nel buffer circolare */
        sem_post(shared.mutex);
        sem_post(shared.nstored);
        /* un altro elemento è disponibile */
    }
    return(NULL);
}
```

Produttore

- Il produttore chiama **sem_wait()** sul semaforo **nempty** per aspettare che ci sia spazio disponibile per un altro elemento nel buffer
 - La prima volta che è eseguita questa istruzione il valore del semaforo andrà da **NBUFF** a **NBUFF-1**
- Prima di memorizzare un nuovo elemento nel buffer, il produttore deve ottenere il semaforo **mutex**
 - Dopo aver memorizzato l'elemento nel buffer, il semaforo **mutex** è rilasciato (il valore va da 0 a 1), e viene fatto un post al semaforo **nstored**. La prima volta che è eseguita questa parte, il valore di **nstored** va da 0 a 1

Consumatore

```
void *consume(void *arg)
{
    int          i;

    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nstored);
        /* attende almeno un elemento */
        sem_wait(shared.mutex);
        if (shared.buff[i % NBUFF] != i)
            printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
        sem_post(shared.mutex);
        sem_post(shared.nempty); /* un altro posto libero */
    }
    return (NULL);
}
```

Consumatore

- Quando il valore del semaforo *nstored* è maggiore di 0, sono disponibili diversi valori da elaborare
- Il consumatore prende un elemento dal buffer e verifica che il valore sia corretto, proteggendo l'accesso al buffer con il semaforo *mutex*
- Il consumatore poi fa un post al semaforo *nempty* comunicando al produttore che c'è un altro posto vuoto

Deadlock

- Se scambiamo l'ordine delle due chiamate a `sem_wait()` nella funzione `consume`, assumendo che il produttore inizi per primo
 - esso memorizza `NBUFF` elementi nel buffer
 - decrementa il valore del semaforo `nempty` da `NBUFF` a `0` e incrementa il valore del semaforo `nstored` da `0` a `NBUFF`
 - A quel punto il produttore si blocca nella chiamata a `sem_wait(shared.nempty)` poiché il buffer è pieno e non ci sono locazioni libere per un altro elemento
- Il consumatore inizia e verifica i primi `NBUFF` elementi del buffer
 - Decrementa il valore del semaforo `nstored` da `NBUFF` a `0` e incrementa il valore del semaforo `nempty` da `0` a `NBUFF`
 - Il consumatore poi si blocca nella chiamata a `sem_wait(shared.nstored)` dopo aver chiamato `sem_wait(shared.mutex)`
- Il produttore può riprendere poiché il valore di `nempty` è maggiore di `0`, ma il produttore poi chiama `sem_wait(shared.mutex)` e si blocca
- Abbiamo così un *deadlock*: il produttore aspetta il semaforo `mutex`, ma il consumatore occupa questo semaforo ed aspetta il semaforo `nstored`. Ma il produttore non può fare un post al semaforo `nstored` fino a che non ottiene il semaforo `mutex`

Semafori Posix basati su memoria

sem_init e sem_destroy

- Fin qui abbiamo considerato i semafori con nome
 - Identificati da un nome che referencia un file nel filesystem
- Posix fornisce anche i semafori basati su memoria in cui l'applicazione alloca la memoria per il semaforo che successivamente il sistema provvede ad inizializzare

sem_init e sem_destroy

```
#include<semaphore.h>
```

```
int sem_init(sem_t *sem, int shared, unsigned int value);
```

```
int sem_destroy(sem_t *sem);
```

- Un semaforo basato su memoria è inizializzato con `sem_init()` L'argomento `sem` punta ad una variabile `sem_t` che l'applicazione deve allocare
 - Se `shared` è 0, allora il semaforo è condiviso tra i thread del processo, altrimenti il semaforo è condiviso tra processi
 - Quando `shared` è diverso da 0, il semaforo deve essere memorizzato in qualche tipo di memoria condivisa che è accessibile a tutti i processi che useranno il semaforo
 - `value` è il valore iniziale del semaforo
- Una volta finito, il semaforo è deallocato con `sem_destroy()`

Esempio: produttore-consumatore

- Come esempio di uso di semafori basati su memoria, consideriamo il problema Produttore - Consumatore con più produttori ed un consumatore

Globali

```
#include      ...

#define      NBUFF      10
#define      MAXNTHREADS      100

int      nitems, nproducers;
/* sola lettura per produttori e consumatore */

struct {      /* dati condivisi tra prods. e cons. */
    int buff[NBUFF];
    int nput;
    int nputval;
    sem_t mutex, nempty, nstored; // semafori non puntatori
} shared;

void      *produce(void *), *consume(void *);
```

Globali

- **nitems** è il numero totale di elementi da produrre per tutti i produttori e **nproducers** è il numero di thread produttori. Entrambi sono impostati da linea di comando
- Ci sono due nuove variabili nella struttura shared: **nput**, l'indice della prossima entrata del buffer in cui memorizzare (modulo NBUFF) e **nputval**, il prossimo valore da memorizzare nel buffer. Queste variabili servono per sincronizzare i thread produttori multipli

main ...

```
int
main(int argc, char **argv)
{
    int            i, count[MAXNTHREADS];
    pthread_t      tid_produce[MAXNTHREADS], tid_consume;

    if (argc != 3)
        {printf("usage: prodcons3 <#items>
<#producers>");exit(-1);}
    nitems = atoi(argv[1]);
    nproducers = MIN(atoi(argv[2]), MAXNTHREADS);
        /* inizializza tre semafori */
    sem_init(&shared.mutex, 0, 1);
    sem_init(&shared.nempty, 0, NBUFF);
    sem_init(&shared.nstored, 0, 0);
        /* crea tutti i produttori ed un consumatore */
    for (i = 0; i < nproducers; i++) {
        count[i] = 0;
        pthread_create(&tid_produce[i], NULL, produce,
&count[i]);
    }
    pthread_create(&tid_consume, NULL, consume, NULL);
}
```

... main

```
/* aspetta tutti i produttori ed il consumatore*/  
for (i = 0; i < nproducers; i++) {  
    pthread_join(tid_produce[i], NULL);  
    printf("count[%d] = %d\n", i, count[i]);  
}  
pthread_join(tid_consume, NULL);  
  
sem_destroy(&shared.mutex);  
sem_destroy(&shared.nempty);  
sem_destroy(&shared.nstored);  
exit(0);  
}
```

main

- Gli argomenti da linea di comando specificano il numero di elementi da memorizzare nel buffer ed il numero di thread produttori da creare
- Inizializzati i semafori, sono creati tutti i thread produttori ed il thread consumatore
- Aspettiamo poi che tutti i thread terminino

Produttori

```
void *produce(void *arg)
{
    for ( ; ; ) {
        sem_wait(&shared.nempty);
        /* aspetta almeno una locazione libera */
        sem_wait(&shared.mutex);

        if (shared.nput >= nitems) {
            sem_post(&shared.nempty);
            sem_post(&shared.mutex);
            return(NULL);          /* tutto prodotto */
        }

        shared.buff[shared.nput % NBUFF] = shared.nputval;
        shared.nput++;
        shared.nputval++;

        sem_post(&shared.mutex);
        sem_post(&shared.nstored);
        /* un altro elemento memorizzato */
        *((int *) arg) += 1;
    }
}
```


Produttori

- Il ciclo termina quando gli **nitem** valori sono stati posti nel buffer da tutti i thread
- Osserviamo che i produttori multipli possono acquisire il semaforo **nempty** allo stesso tempo, ma solo un produttore può acquisire il semaforo **mutex** per proteggere le variabili **nput** e **nputval**

Consumatore

```
void *consume(void *arg)
{
    int      i;
    for (i = 0; i < nitems; i++) {
        sem_wait(&shared.nstored);
        /* attende almeno un elemento memorizzato */
        sem_wait(&shared.mutex);

        if (shared.buff[i % NBUFF] != i)
            printf("error:buff[%d]=%d\n",i,shared.buff[i% NBUFF]);
        sem_post(&shared.mutex);
        sem_post(&shared.nempty);
        /* un altro posto libero */
    }
    return(NULL);
}
```

Consumatore

- Verifica che ciascuna entrata del buffer sia corretta, mentre stampa un messaggio in caso contrario

Condivisione di semafori tra processi

- Le regole per condividere semafori basati su memoria sono semplici: il semaforo stesso (il tipo `sem_t`) risiede in un'area di memoria che deve essere condivisa da tutti i processi che vogliono condividere il semaforo, ed il secondo argomento di `sem_init()` deve essere 1
- Per i semafori con nome, processi differenti possono sempre far riferimento allo stesso semaforo facendo sì che ogni chiamata a `sem_open()` specifichi lo stesso nome

Condivisione di semafori tra Processi

- I semafori basati su memoria hanno persistenza di processo, ma la reale persistenza dipende dal tipo di memoria in cui il semaforo è memorizzato
 - Il semaforo esiste fin quando la memoria che lo contiene è valida
 - Se il semaforo di memoria è condiviso tra thread di un processo singolo (l'argomento *shared* di `sem_init()` è 0), allora il semaforo ha persistenza di processo e scompare quando il processo termina
 - Se invece è condiviso tra processi differenti (*shared* vale 1 in `sem_init()`), allora il semaforo deve essere memorizzato in memoria condivisa e resta disponibile fino a che la porzione di memoria condivisa rimane disponibile

Condivisione di semafori tra Processi

- Poniamo l'accento sul fatto che il seguente codice non può funzionare:

```
sem_t mysem;  
sem_init(&mysem, 1, 0);  
if (fork() == 0) {  
    ...  
    sem_post(&mysem);  
}  
sem_wait(&mysem);
```

- Il semaforo *mysem*, infatti, non è in memoria condivisa. La memoria non è condivisa tra padre e figlio attraverso una fork. Il figlio inizia con una copia della memoria del padre, ma questo è diverso dal condividere la memoria

Un altro (semplice) esempio

- I semafori possono essere usati per risolvere diversi problemi di sincronizzazione
- Si considerino, per esempio, due processi in esecuzione concorrente:
 - *P1* esegue un'istruzione *S1* e *P2* esegue un'istruzione *S2*
 - Si supponga di voler eseguire *S2* solo dopo che *S1* è terminata (indipendentemente dallo scheduling dei due processi)
- Questo schema si può realizzare facendo condividere a *P1* e *P2* un semaforo comune, *sincronizzazione*, inizializzato a 0

Un altro (semplice) esempio

- Nel processo **P1** si inseriscono le istruzioni

...

S1;

sem_post(sincronizzazione);

- E nel processo **P2** le istruzioni:

...

sem_wait(sincronizzazione);

S2;


```

#include ...
int main(int arg, char **argv){
    sem_t *sincronizzazione;
    int pid;
    sincronizzazione = sem_open("/test",O_CREAT,0666,0);
    pid = fork();
    if (pid==0){
        sem_wait(sincronizzazione);
        printf("di Sistemi Operativi\n");  /* Istruzione S2*/
        exit(0);
    }
    else {
        printf("Laboratorio ");             /* Istruzione S1 */
        sem_post(sincronizzazione);
        fflush(NULL);
        wait(NULL);
        sem_unlink("/test");
        exit(0);
    }
}

```

Esercizio

- Realizzare un programma C e Posix sotto Linux che con l'uso dei semafori Posix sincronizzi un processo padre ed un processo figlio che scrivono e leggono, rispettivamente, un numero intero alla volta (da 1 a n, dove n è passato da riga di comando) nella prima posizione di un file temporaneo opportunamente creato